Static Slicing of Threaded Programs

Jens Krinke
krinke@ips.cs.tu-bs.de
TU Braunschweig
Abteilung Softwaretechnologie

Abstract

Static program slicing is an established method for analyzing sequential programs, especially for program understanding, debugging and testing. Until now, there was no slicing method for threaded programs which handles interference correctly. We present such a method which also calculates more precise static slices. This paper extends the well known structures of the control flow graph and the program dependence graph for threaded programs with interference. This new technique does not require serialization of threaded programs.

1 Introduction

Static program slicing [19] is an established method for analyzing sequential programs, especially for program understanding, debugging and testing. But today even small programs use parallelism and a method to slice such programs is required. *Dynamic* slicing of threaded (or *concurrent*) programs has been researched by several authors. But only one approach for *static* slicing of threaded programs is known to us [1, 2]. A drawback of this approach is that the calculated slices are not precise enough, because it does not handle *interference*. Interference is data flow which is introduced through use of variables which are common to parallel executing statements. We approach that problem and present a more precise algorithm for static slicing of threaded programs with interference.

The analysis of programs where some statements may explicitly be executed in parallel is not new. The *static* analysis of these programs is complicated, because the execution order of parallel executed statements is *dynamic*. Testing and debugging of threaded programs have increased complexity: they might produce different behavior even with the same input. The nondeterministic behavior of a program is hard to understand and finding harmful nondeterministic behavior is even harder. Therefore, supporting tools are required. Unfortunately, most tools for sequential programs are not applicable to threaded programs, as they cannot cope with the nondeterministic execution order of statements. One simple way to circumvent these problems is to simulate these programs through *sequentialized* or *serialized* programs [18]. These are “product” programs, in which every possible execution order of statements is modeled through a path where the statements are executed sequentially. This may lead to exponential code explosion, which is often unacceptable for analysis. Therefore, special representations of parallel programs have been developed.

In the following sections we will first introduce our notation of threaded programs and show how to extend control flow graphs (CFGs) and program dependence graphs (PDGs) to threaded PDGs, which are our base for slicing. The problem of static slicing threaded programs is explained in section 4, where we also present an algorithm to slice these programs. The last two sections present some related work and discuss the conclusions and further work.

2 The threaded CFG

A common way to represent procedures of a program are *control flow graphs* (CFG). A CFG is a directed graph $G = (N, E, s, e)$ with node set N and edge set E. The statements and predicates are represented by nodes $n \in N$ and the flow of control between statements is represented by edges $(n, m) \in E$ and written as $n \to m$. Two special nodes s and e are distinguished, the START node s and the EXIT node e which represent the beginning and the end of the procedure. Node s does not have predecessors and node e does not have successors. The variables which are referenced at node[1] n are denoted by $ref(n)$, the variables which are defined (or assigned) at n are denoted by $def(n)$.

[1]In the rest of this paper we will use “node” and “statement” interchangeable, as they are bijectively mapped

A *path* in G is a sequence $P = \langle n_1, \ldots, n_k \rangle$ where $n_i \rightarrow n_{i+1}$ for all $1 \leq i < k$. A node p is *reachable* ($q \rightarrow^\star p$) from another node q, if there is a path $\langle q, \ldots, p \rangle$ in G. i. e. "$\rightarrow^\star$" is the transitive, reflexive closure of "$\rightarrow$". We assume that every path in a CFG is a possible execution order of the statements of the program. If we pick some statements out of this sequence they are a *witness* of a possible execution.

Definition 2.1 We call a sequence $\langle n_1, \ldots, n_k \rangle$ of nodes a *witness*, iff $n_i \rightarrow^\star n_{i+1}$ for all $1 \leq i < k$.

This means that a sequence of nodes is a witness, if all nodes are part of a path through the CFG in the same order as in the sequence. Every path is a witness of itself.

A *thread* is a part of a program which must be executed on a single processor. Threads may be executed in parallel on different processors or interleaved on a single processor. In our model we assume that threads are created through `cobegin`/`coend` statements and that they are properly synchronized on statement level. Let the set of threads be $\Theta = \{\theta_0, \theta_1, \ldots, \theta_n\}, n = |\Theta| + 1$. For simplicity we consider the main program as a thread θ_0.

A sample program with two threads is shown in Figure 1. Thread θ_1 is the block of statements S_3, S_4 and S_5 and the other thread θ_2 is the block with S_6 and S_7. S_1, S_2 and S_8 are part of the main program θ_0.

A *threaded* CFG (tCFG) extends the CFG with two special nodes COSTART and COEXIT which represent the `cobegin` and `coend` statements. The enclosed threads are handled like complete procedures and will be represented by whole CFGs, which are embedded in the surrounding CFG. The START and EXIT nodes of these CFGs are connected to the COSTART and COEXIT nodes with special *parallel* flow edges. We will distinguish the edges through $p \xrightarrow{cf} q$ for a sequential control flow edge between nodes p and q and $p \xrightarrow{pf} q$ for a parallel flow edge. Figure 2 shows the tCFG for the example program of Figure 1.

$\theta(p)$ is a function which returns for every node p its innermost enclosing thread. In the example we have $\theta(S_2) = \theta_0$, $\theta(S_4) = \theta_1$ and $\theta(S_6) = \theta_2$. $\overline{\Theta}(p)$ is a function that returns for every node p the set of threads which cannot execute parallel to the execution of p, e. g. $\overline{\Theta}(S_4) = \emptyset$ or $\overline{\Theta}(S_2) = \{\theta_1, \theta_2\}$.

The definition of witnesses in CFGs may also be applied to tCFGs. But this does not take the possible interleaving of nodes into account and we have to extend the definition:

Definition 2.2 A sequence $l = \langle n_1, \ldots, n_k \rangle$ of nodes is a *threaded witness* in a tCFG, iff

$$\forall_{t \in \Theta} : l\,\big|_t = \langle m_1, \ldots, m_j \rangle \Rightarrow \forall_{i=1}^{j-1} : m_i \overset{cf,pf}{\longrightarrow^\star} m_{i+1}$$

where $l\,|_t$ is the subsequence of $l = \langle n_1, \ldots, n_k \rangle$ in which all nodes n_i with $\theta(n_i) \neq t$ have been removed.

```
S1:   x = ...;
S2:   i = 1;
      cobegin {
        if (x>0) {
S3:       x = -x;
S4:       i = i+1;
        } else {
S5:       i = i+1;
        }
      }{
S6:     i = i+1;
S7:     z = y;
      } coend;
S8:   ... = i;
```

Figure 1: A threaded program

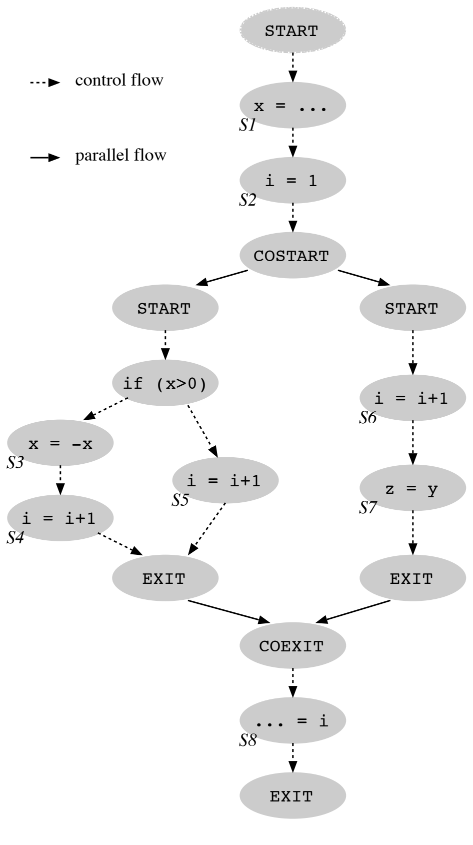

Figure 2: A threaded CFG

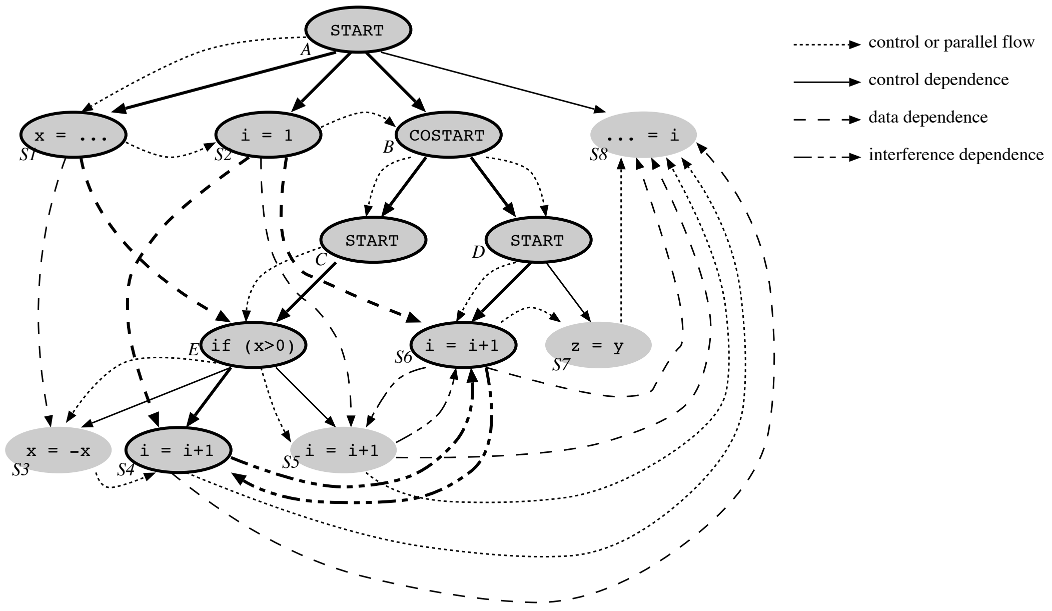

Figure 3: A threaded PDG

Intuitively, a threaded witness can be interpreted as a witness in the sequentialized CFG. This definition assures that a sequence of nodes, which are part of different threads, is a witness in each of the different threads. Every ordinary witness in the tCFG is automatically a threaded witness. In our example of Figure 2, $\langle S_1, S_4, S_6\rangle$ and $\langle S_1, S_2, S_8\rangle$ are threaded witnesses and $\langle S_5, S_6, S_4\rangle$ or $\langle S_1, S_4, S_5\rangle$ are not. The sequence $\langle S_1, S_2, S_8\rangle$ is also an ordinary witness but the sequence $\langle S_1, S_4, S_6\rangle$ is not.

3 The threaded PDG

A *program dependence graph* [5] is a transformation of a CFG, where the control flow edges have been removed and two other kinds of edges have been inserted: *control dependence* and *data dependence* edges.

Definition 3.1 A node j is called *data dependent* on node i, if

1. there is a path P from i to j in the CFG ($i \rightarrow^\star j$).
2. there is a variable v, with $v \in \mathit{def}(i)$ and $v \in \mathit{ref}(j)$
3. for all nodes $k \neq i$ of path $P \Rightarrow v \notin \mathit{def}(k)$.

Node j is called a *postdominator* of Node i, if any path from i to EXIT must go through j. A node i is called a *predominator* of j if any path from START to j must go through i. In typical programs, statements in loop bodies are predominated by the loop entry and postdominated by the loop exit.

Definition 3.2 A node j is called *(direct) control dependent* on node i, if

1. there is a path P from i to j in the CFG ($i \rightarrow^\star j$).
2. j is a postdominator for every node in P except i
3. j is not a postdominator for i.

The PDG consists of the nodes of the CFG and control dependence edges $p \xrightarrow{cd} q$ for nodes q which are control dependent on nodes p, and data dependence edges $p \xrightarrow{dd} q$ for nodes q which are data dependent on nodes p.

Definition 3.3 A node j is called *transitive dependent* on node i, if

1. there is a path $P = \langle i = n_1, \ldots, n_l = j\rangle$ where every n_{k+1} is control or data dependent on n_k
2. P is a witness in the CFG

Note that the composition of control and data dependence is always transitive: A dependence between x and y and a dependence between y and z are implying a path between x and z from the definition of control and data dependence.

There have been some attempts to define threaded variants of PDGs. To the best of our knowledge none of these explicitly represents the dependences which result from *interference*. Interference occurs if a variable is defined in one thread and referenced in another parallel executing thread. In the example of Figure 1 we have an interference for the variable `i` between θ_1 and θ_2. The value of `i` at statement

```
S1:  i = 1;
     cobegin {
       while (z>0) {
         cobegin {
S2:        x = i;
         }{
S3:        y = x;
         } coend;
       }
     }{
S4:    z = y;
     } coend;
S5:  x = z;
```

Figure 4: A program with nested threads

S_6 may be the value computed at S_2, S_4 or S_5. The value of `i` at statement S_8 may be the value computed at S_4, S_5 or S_6. However, if the statements S_4, S_5 and S_6 are properly synchronized, the value of `i` will always be 3.

Definition 3.4 A node j is called *interference dependent* on node i, if

1. $\theta(i) \neq \theta(j)$ and $\theta(j) \notin \overline{\Theta}(i)$, i. e. $\theta(i)$ and $\theta(j)$ may potentially be executed in parallel,
2. there is a variable v, such that $v \in def(i)$ and $v \in ref(j)$

Dependences between threads which are not executed in parallel are ordinary data dependences.

The dependences introduced by interference cannot be handled with normal data dependence as normal dependence is transitive and interference dependence is not. The transitivity of the data and control dependence results from their definitions, where a sequential path between the dependent nodes is demanded. The composition of paths in the CFG always results in a path again.

Interference dependence is not transitive: If a statement x is interference dependent on a statement y, which is interference dependent on z, then x is only dependent on z iff there is a possible execution where these three statement are executed one after another: The sequence $\langle x, y, z\rangle$ of the three statements has to be a threaded witness in the tCFG. In the example of Figure 3 statement S_4 is interference dependent on statement S_6, which in turn is interference dependent on statement S_5. However, there is no possible execution where S_4 is executed after S_5 and thus S_4 cannot be interference dependent on S_5, $\langle S_5, S_6, S_4\rangle$ is no threaded witness.

A *threaded program dependence graph* (tPDG) consists of the nodes and the edges of the tCFG with the addition of control, data and interference dependence edges. In contrast to the standard PDG, where the control flow edges have been removed, we need the control and parallel flow edges for reasons we will explain later. As usual, the EXIT and COEXIT nodes can be removed, if the control and parallel flow edges are adapted accordingly. The tPDG of the example is shown in Figure 3.

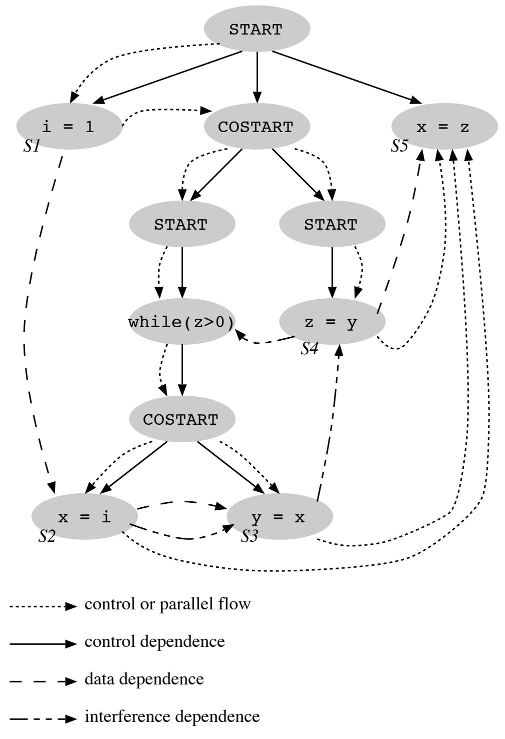

Figure 5: The tPDG of Figure 4

More complicated structures like loops or nested threads may be handled in the same way. An example is shown in Figure 4. In the tPDG in Figure 5 there is both a data and an interference dependence edge between statement S_2 and S_3. Both statements and their threads may be executed in parallel (therefore the interference dependence). The statements and their threads may also be executed sequentially through different iterations of the enclosing loop.

The technique to calculate the edges is beyond the scope of the papers, they can be calculated with standard algorithms [8]. A simple version would assume the existence of a boolean function *parallel*(i, j) which returns *true* if it is possible for nodes i and j to execute in parallel (see [12] for an overview of ways to calculate this function). An interference dependence edge $i \xrightarrow{id} j$ will be inserted for all (i, j) if there is a variable v which is defined at i, referenced at j and *parallel*(i, j) is true.

4 Slicing the tPDG

Slicing on the PDG of sequential programs is a simple graph reachability problem [14], because control and data dependence is transitive.

Definition 4.1 The (*backward*) *slice* $S(p)$ of a (sequential) PDG at node p consists of all nodes on which p (transitively) depends:

$$S(p) = \{q | q \rightarrow^{\star} p\}$$

The node p is called the *slicing criterion*.

This definition may easily implemented through a graph reachability algorithm. As interference dependence is not transitive, this definition of a slice for PDGs is not valid for tPDGs and hence the standard algorithms are not really applicable.[2]

The basic idea of our approach stems from a simple observation: Because every path in the PDG is a witness in the corresponding CFG, every node p which is reachable from a node q in the PDG, is also reachable from q in the corresponding CFG. This does not hold for the threaded variants. The definition of a slice in the tPDG establishes a similar property, because it demands that the tPDG contains a threaded witness between every node in the slice and the slicing criterion.

Definition 4.2 The (*backward*) *slice* $S_\theta(p)$ of a tPDG at a node p consists of all nodes q on which p transitively depends:

$$\begin{aligned} S_\theta(p) = \{q \quad | \quad & P = \langle n_1, \ldots, n_k \rangle, \\ & q = n_1 \xrightarrow{d_1} \ldots \xrightarrow{d_{k-1}} n_k = p, \\ & d_i \in \{cd, dd, id\},\ 1 \le i < k. \\ & \text{and } P \text{ is a threaded witness} \\ & \text{in the tCFG}\} \end{aligned}$$

A slice from the statement S_4 of the example program in Figure 1 is shown in Figure 3 as framed nodes. The responsible edges are drawn in a thicker style. Note that there are interference edges between statement S_6 and S_5 which does not force the inclusion of statement S_5 into the slice because S_4 is not reachable from S_5 in the tCFG. The standard slicing algorithm would include the statement S_5 into the slice, which is, albeit correct, to inaccurate.

The algorithm to slice sequential programs is a simple reachability algorithm. However, it is not easy to transform the definition of a threaded slice into an algorithm because the calculation of threaded witnesses would be too costly. Therefore we present a different slicing algorithm in Figure 6. Its basic idea is the coding of possible states of execution in all threads in tuples $(t_0, t_1, \ldots, t_{|\Theta|-1})$, where the t_i are nodes in the tPDG with $\theta(t_i) = \theta_i$. The value t_i represents a node which has not yet been reached by the execution of thread θ_i and it is still possible to reach node t_i. A value of $\perp$ does not restrict the state of execution. This is used to keep track of the nodes p where a thread has been left through following an interference edge. If we follow another interference edge back into the thread at node q, we are able to check that p is reachable from q. This assures that paths over interference edges are always threaded witnesses in the tCFG. This is the reason why we have to keep the control and parallel flow edges in the tPDG.

[2]The "classical" definition of a slice is any subset of a program that does not change the behaviour in respect to the criterion: a program is a correct slice of itself. Therefore, if interference is modelled with normal data dependence, the resulting slices are correct but unprecise.

```
**Input:** the slicing criterion $s$, a node of the tPDG
**Output:** the slice $S$, a set of nodes of the tPDG
*Initialize the worklist with an initial state tuple:*
$C = (s, (t_0, \ldots, t_{|\Theta|})) \; \Big| \; t_i = \begin{cases} s & \text{if } \theta(s) = \theta_i \\ \perp & \text{else} \end{cases}$
worklist $w = \{C\}$
slice $S = \{s\}$
**repeat**
    remove the next element $c = (x, T)$ from $w$
    *Examine all reaching edges:*
    **for all** edges $e = y \xrightarrow{cd,dd} x$ **do**
        $T' = [y/_{\theta(y)}]T$
        **if** $\theta(y) \neq \theta(x)$ **then**
            *Normal dependence between threads:*
            *reset the exited threads*
            *(which cannot execute parallel to y)*
            **for all** $t \in \overline{\Theta}(y)$ **do**
                $T' = [\perp/_t]T'$
        $c' = (y, T')$
        **if** $c'$ has not been already calculated **then**
            mark $c'$ as calculated
            $w = w \cup \{c'\}$
            $S = S \cup \{y\}$
    **for all** edges $e = y \xrightarrow{id} x$ **do**
        $t = T[\theta(y)]$
        **if** $t = \perp$ **or** $y \xrightarrow{cf,pf}{}^{\star} t \neq y$ **then**
            *The inclusion of the edge still results*
            *in a threaded witness*
            $c' = (y, [y/_{\theta(y)}]T)$
            **if** $c'$ has not been already calculated **then**
                mark $c'$ as calculated
                $w = w \cup \{c'\}$
                $S = S \cup \{y\}$
**until** worklist $w$ is empty.
```

Figure 6: Slicing algorithm

We denote the extraction of the ith element t_i in a tuple $T = (t_0, t_1, \ldots, t_n)$ with $T[i]$. The substitution of the ith element t_i in a tuple $T = (t_0, t_1, \ldots, t_n)$ with a value x will be denoted as $[x/_i](T)$.

The algorithm keeps a worklist of pairs of nodes and state tuples which have to be examined. Every edge reaching the node is examined and is handled dependently of its type. In case of a control or data dependence edge, a new pair consisting of the source node and the modified state tuple is inserted into the worklist. The new state tuple has the source node as the actual state of its thread. If the edge crosses threads, the state of the left threads are resetted. In the other case its an interference dependence edge. It may only be considered if the state node of the source node thread is reachable from the source node in the tCFG (all examined paths are still threaded witnesses). Then, the new pair with the updated state tuple is inserted into the worklist. The resulting slice is the set of nodes which is constructed of the first elements of the inserted pairs.

In the following we will demonstrate an application of the algorithm to calculate a backward slice for node S_4. The worklist w is initialized with the element $(S_4, (\perp, S_4, \perp))$. This element is immediately removed from the worklist and all edges reaching S_4 are examined. The edge $E \xrightarrow{cd} S_4$ does not cross threads and the state of the thread $\theta(S_4) = \theta(E)$ is updated before the created element $(E, (\perp, E, \perp))$ is inserted into the worklist. The edge $S_2 \xrightarrow{dd} S_4$ does cross threads and the state of the exited threads is reset. This creates a new element $(S_2, (S_2, \perp, \perp))$. The edge $S_6 \xrightarrow{id} S_4$ creates $(S_6, (\perp, S_4, S_6))$, because the state of $\theta(S_6)$ is $\perp$. Let us step forward in the calculation and assume the worklist is $\{(S_6, (\perp, S_4, S_6)), (C, (\perp, C, \perp)), \ldots\}$. There are four edges reaching S_6:

1. $S_2 \xrightarrow{dd} S_6$ crosses threads and creates the element $(S_2, (S_2, \perp, \perp))$. As this element has already been visited, it is not inserted into the worklist again.

2. $D \xrightarrow{cd} S_6$ does not cross threads and inserts the element $(D, (\perp, S_4, D))$ into the worklist.

3. $S_5 \xrightarrow{id} S_6$: as $(\perp, S_4, S_6)[\theta(S_5)] = S_4$ and the condition $S_5 \xrightarrow{cf,pf}{}^{\star} S_4$ is not fulfilled, this edge has to be ignored.

4. $S_4 \xrightarrow{id} S_6$: the condition $T[\theta(S_4)] \neq S_4$ cannot be fulfilled and this edge has to be ignored.

In the third step, the edge has to be ignored because it would destroy the property that every node in the slice is part of a threaded witness. The condition which is not fulfillable in step four may be relaxed if we drop our assumption that the program is properly synchronized on statement level. The remaining calculations are presented in Figure 7.

$w : \{(S_4, (\perp, S_4, \perp))\}$
$E \xrightarrow{cd} S_4 \;\Rightarrow\; (E, (\perp, E, \perp))$
$S_2 \xrightarrow{dd} S_4 \;\Rightarrow\; (S_2, (S_2, \perp, \perp))$
$S_6 \xrightarrow{id} S_4 \;\Rightarrow\; (S_6, (\perp, S_4, S_6))$
$w : \{(E, (\perp, E, \perp)), (S_2, (S_2, \perp, \perp)), (S_6, (\perp, S_4, S_6))\}$
$C \xrightarrow{cd} E \;\Rightarrow\; (C, (\perp, C, \perp))$
$S_1 \xrightarrow{dd} E \;\Rightarrow\; (S_1, (S_1, \perp, \perp))$
$w : \{(S_2, (S_2, \perp, \perp)), (S_6, (\perp, S_4, S_6)), (C, (\perp, C, \perp)), (S_1, (S_1, \perp, \perp))\}$
$A \xrightarrow{cd} S_2 \;\Rightarrow\; (A, (A, \perp, \perp))$
$w : \{(S_6, (\perp, S_4, S_6)), (C, (\perp, C, \perp)), (S_1, (S_1, \perp, \perp)), (A, (A, \perp, \perp))\}$
$S_2 \xrightarrow{dd} S_6 \;\Rightarrow\; (S_2, (S_2, \perp, \perp))$ already visited
$D \xrightarrow{cd} S_6 \;\Rightarrow\; (D, (\perp, S_4, D))$
$S_5 \xrightarrow{id} S_6 \;\Rightarrow\; S_5 \xrightarrow{cf,pf}{}^{\star} S_4$ is not fulfilled $(T[\theta(S_5)] = S_4)$
$S_4 \xrightarrow{id} S_6 \;\Rightarrow\; T[\theta(S_4)] \neq S_4$ is not fulfilled $(T[\theta(S_4)] = S_4)$
$w : \{(C, (\perp, C, \perp)), (S_1, (S_1, \perp, \perp)), (A, (A, \perp, \perp)), (D, (\perp, S_4, D))\}$
$B \xrightarrow{cd} C \;\Rightarrow\; (B, (B, \perp, \perp))$
$w : \{(S_1, (S_1, \perp, \perp)), (A, (A, \perp, \perp)), (D, (\perp, S_4, D)), (B, (B, \perp, \perp))\}$
$A \xrightarrow{cd} S_1 \;\Rightarrow\; (A, (A, \perp, \perp))$ already in worklist
$w : \{(A, (A, \perp, \perp)), (D, (\perp, S_4, D)), (B, (B, \perp, \perp))\}$
no edge reaching A exists
$w : \{(D, (\perp, S_4, D)), (B, (B, \perp, \perp))\}$
$B \xrightarrow{cd} D \;\Rightarrow\; (B, (B, \perp, \perp))$ already in worklist
$w : \{(B, (B, \perp, \perp))\}$
$A \xrightarrow{cd} B \;\Rightarrow\; (A, (A, \perp, \perp))$ already visited

$$\Rightarrow \quad S_\theta(S_4) = \{S_4, E, S_2, S_6, C, S_1, A, D, B\}$$

Figure 7: Calculation of $S_\theta(S_4)$

If we assume that the analyzed programs has no threads, $\Theta = \{\theta_0\}$, then this algorithm is similar to the sequential slicing algorithm. In that case, the second iteration over all interference dependence edges will not be executed and the worklist will only contain tuples of the form $(n, (n))$, where n is a node of the PDG. Hence the standard slicing algorithm on PDGs is a special case of our algorithm, which has the same time and space complexity for the unthreaded case.

In the threaded case the reachability $y \xrightarrow{cf,pf}{}^{\star} x$ has to be calculated iteratively. This determines the worst case for time complexity in the number of interference edges: the traversal of these edges might force another visit of all nodes that may reach the source of the edge. Therefore, the worst case is exponential in the number of interference dependence edges. We believe that the number of interference dependence edges will be very small in every program, as interference is error prone, hard to understand and to debug. The required calculation time will be much less than the time required to analyze serialized programs.

5 Related work

There are many variations of the program dependence graph for threaded programs like parallel program graphs [15, 2, 1, 4]. However, most of them are unusable for static slicing.

Dynamic slicing of threaded or *concurrent* programs has been approached by different authors [4, 13, 3, 9] and is surveyed in [17].

The only other approach to static slicing of threaded programs known to the author is the work of Cheng [1, 2]. He introduces some dependences, which are even more specialized than our interference dependence. These are needed for a variant of the PDG, the *program dependence net* (PDN). His *selection* dependence is a special kind of control dependence and his *synchronization* dependence is a mixture of control and data dependence. Our interference dependence is most similar to his *communication* dependence, where dependence is introduced through explicit interprocess communication. Although our tPDG is not mappable to his PDN and vice versa, both graphs are similar in the number of nodes and edges.

Cheng defines slices simply based on graph reachability. The resulting slices are not precise, as they do not take into account that dependences between parallel executed statements are not transitive. Therefore, the integration of his technique of slicing threaded programs into slicing threaded object oriented programs [20] has the same problem.

6 Conclusions and further work

We have presented extended versions of the control flow and program dependence graphs for threaded programs, called the *threaded control flow graph* and *threaded program dependence graph*. The tCFG is similar to other extensions of the CFG for threaded programs. The tPDG is new, as it captures the interference in threaded programs. With the tPDG we are able to calculate better static slices of threaded programs than previous approaches.

We believe that, as more and more programs are using threads, static slicing of them will become more important. We plan to extend our method to handle

procedures. The presented algorithm works only intraprocedural. However, known techniques [7] for interprocedural slicing can be integrated straightforward.

synchronization. For simplicity, we have assumed implicit synchronization of the analyzed programs. Our plan is to integrate explicit synchronization similar to [2].

different threads. The `cobegin`/`coend` model is not always sufficient to model different types of parallelism. We are planning to extend our technique for different kind of threads like `fork`/`join`.

object orientation. The problem of slicing object oriented programs is orthogonal to slicing threaded programs, the integration of slicing object oriented programs like [11] should be possible, following similar techniques as [20].

Our next goal is the integration of this technique in our slicing tool [6, 16] for sequential standard C programs. As this tool is able to generate and simplify path conditions based on program slices, we will develop new constraints stemming from threaded program for these path conditions to obtain an even better slice accuracy.

Acknowledgments

The author wishes to thank Gregor Snelting, Torsten Robschink and especially Bernd Fischer for their helpful support. This work was funded by the Bundesministerium für Bildung und Forschung, FKZ 01 IS 513 C9.

References

[1] J. Cheng. Slicing concurrent programs. In *Automated and Algorithmic Debugging, 1st Intl. Workshop*, LNCS 749, 1993.

[2] J. Cheng. Dependence analysis of parallel and distributed programs and its applications. In *Intl. Conf. on Advances in Parallel and Distributed Computing*, 1997.

[3] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4), 1991.

[4] E. Duesterwald, R. Gupta, and M. L. Soffa. Distributed slicing and partial re-execution for distributed programs. In *5th Workshop on Languages and Compilers for Parallel Computing*, LNCS 757, 1992.

[5] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3), 1987.

[6] M. Goldapp, U. Grottker, and G. Snelting. Validierung softwaregesteuerter Meßsysteme durch Program Slicing und Constraint Solving. In *Statusseminar des BMBF Softwaretechnologie*, Berlin, 1996.

[7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), 1990.

[8] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programms. *ACM Transactions on Programming Languages and Systems*, 18(3), 1996.

[9] B. Korel and R. Ferguson. Dynamic slicing of distributed programs. *Applied Mathematics and Computer Science*, 2, 1992.

[10] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3), 1988.

[11] L. D. Larsen and M. J. Harrold. Slicing object-oriented software. In *Proc. 18th Intl. Conf. on Software Engineering*, 1996.

[12] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4), 1989.

[13] B. P. Miller and J. D. Choi. A mechanism for efficient debugging of parallel systems. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1988.

[14] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1984.

[15] V. Sarkar and B. Simons. Parallel program graphs and their classification. In *Proc. 6th Workshop on Languages and Compilers for Parallel Computing*, LNCS 768, 1993.

[16] G. Snelting. Combining slicing and constraint solving for validation of measurement software. In *Static Analysis; Third Intl. Symposium*, LNCS 1145, 1996.

[17] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.

[18] N. Uchihira, S. Honiden, and T. Seki. Hypersequential programming. *IEEE Concurrency*, July-September 1997.

[19] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4), 1984.

[20] J. Zhao, J. Cheng, and K. Ushijima. Static slicing of concurrent object-oriented programs. In *Proc. 20th IEEE Annual Intl. Computer Software and Applications Conf.*, 1996.